THE BELL SYSTEM
TECHNICAL JOURNAL

DEVOTED TO THE SCIENTIFIC AND ENGINEERING
ASPECTS OF COMPUTING

Volume 62 December 1983 Number 10, Part 2

Theory of Program Testing—An Overview

By R. E. PRATHER*

(Manuscript received January 18, 1983)

In this paper, we provide a detailed survey of the various approaches to program testing that have been proposed in recent years. Particular attention is given to a discussion of the developing theory of program testing and to the decomposition of the testing problem into the program graph construction, test path selection, and test case generation phases. Examples are included to illustrate the different testing strategies. Comparisons are made from one method to another, all in a uniform terminology and notation, to facilitate an understanding of various combinations of strategies that might lead to a more workable testing methodology.

I. INTRODUCTION

The general goal of software testing is to affirm the quality of a program through systematic exercising of the code in a carefully controlled environment. The execution of a program test scheme should validate an expected prespecified behavior, ideally serving to demonstrate the absence of program errors. Considering the difficulty of obtaining actual proofs of program correctness, program testing

* University of Denver, Colorado.

may he the only effective means for assuring the quality of software systems of nontrivial complexity.

Tbe state of the art in software testing as of a decade ago is broadly surveyed in the hook hy Hetzel,[1] representing an ad hoc approach at hest. During the intervening years, computer programming methodology has made great strides toward improving the quality of our product. And yet, software testing bas remained a kind of "hlack art", only vaguely understood by its practitioners. Happily, tbis situation is changing. The development of the heginnings of a theory of testing are well under way, and the more recent literature shows great promise for hrighter days ahead. Some of tbese ideas are discussed in a new hook hy Myers,[2] and further elaboration can he found in the survey papers by Miller.[3-6]

In this overview, we summarize in detail the more recent literature on software testing and present the more important results in a uniform framework, style, and notation. We hope that this perspective will help to focus attention on the more viahle alternatives and to point the way toward the most promising directions for future research and development.

II. GENERAL THEORY—THE FUNCTIONAL APPROACH

The first attempt to describe a generalized theory of testing is found in tbe work of Goodenougb and Gerhart,[7,8] A related study is that of Hamlet.[9] In the former, a *program* is viewed as a function $F{:}D \rightarrow R$ over an *input domain* D with values in an *output range* R. The program *specification* can also he viewed as a function $G{:}D \rightarrow R$, whether completely specified or not. For testing purposes, we must compare $F(d)$ with $G(d)$ for selected inputs d in D. Though sucb an exhaustive test is not feasihle in general, we say that the program F is *correct* if we bave

$$F(d) = G(d) \text{ (for all } d \text{ in } D),$$

recognizing that this is simply a theoretical notion, one not necessarily capahle of direction verification.

In any practical setting, we will only he ahle to examine the hehavior of the program for a few selected input values. Realizing this, we say that a *test* for the program F is a (finite) suhset T of D. Recalling the 'goal of software testing,' T is said to be an *ideal test* (for F) if

$$\text{success}(T) \Rightarrow \text{correct}(F),$$

i.e., if $F(t) = G(t)$ for t in T implies the same for all t in D. Note that the successful execution of an ideal test would constitute a proof of correctness. Given the difficulty in finding proofs of correctness, however, we should not he surprised to learn that ideal tests, in this

sense, are difficult to discover. (We note that the 'trivial' ideal test, tbe exhaustive one with $T = D$, though easily stated is ordinarily unmanageable in size.)

As a matter of fact, we would prefer not to 'discover' our tests at all, but to have them 'selected' on the basis of some sensible criterion. Formally, a *test selection criterion* (for a program F) is a (true-false) predicate C over the subsets of D. Following Goodenough and Gerhart once again, such a criterion C is *reliable* (for F) if

$$C(T1) \text{ and } C(T2) \Rightarrow \text{success}(T1) = \text{success}(T2),$$

and, on the other hand, C is said to be *valid* (for F) if

$$\sim\text{correct}(F) \Rightarrow \sim\text{success}(T)$$

for some T satisfying $C(T)$. In general, reliability refers to the consistency with which results will be produced within the selection criterion, whereas validity refers to the ability to produce meaningful results, regardless of their consistency.

It is clear that these notions of reliability and consistency are quite strong. Perhaps the most convincing statement to this effect is given by the following:

Theorem (Goodenough and Gerhart): If C is reliable and valid, then C(T) implies that T is an ideal test.

On the other hand, Weyuker and Ostrand[10] have argued that tbese notions are not strong enough, referring as tbey do to a particular program. If the same ideas are extended, however, so as to apply "uniformly" over all programs F, then one obtains the following:

Theorem (Weyuker and Ostrand): If C is uniformly reliable and uniformly valid, then C(T) implies that T = D, i.e., T is an exhaustive test.

Surely tbis carries the original ideas too far. And in fact, the tbeorem can be understood to say, "If nothing is known about the errors in the program, a test criterion is guaranteed ideal (in the sense of Goodenough and Gerhart) if and only if it selects tbe entire input domain." What is probably needed to arrive at a more practical alternative is a weakening of the Goodenough and Gerhart theory. This is the general thrust of Hamlet's work, but results along these lines thus far are less than satisfactory, showing perhaps more promise toward applications to program maintenance than to testing. The interested reader should consult Ref. 9 for details.

A test selection criterion C should outline the properties of a program that must be exercised to constitute a "tborough" test, ideally one whose successful execution implies an error-free program. Following Goodenough and Gerhart once again, we may suppose that C is described as a finite set $\{c\}$ of *test predicates* (i.e., logical conditions on

the input data), and we then choose T subject to the condition(s):

$$C(T) \Leftrightarrow \begin{array}{l}\text{for all c in } C\text{, there is } t \text{ in } T \text{ with c}(t)\\ \text{for all } t \text{ in } T\text{, there is c in } C \text{ with c}(t).\end{array} \qquad (*)$$

In words, every test predicate belonging to C should be satisfied by at least one test datum t in T, and conversely, every t in T must satisfy at least one test predicate.

It is suggested that the test predicates be derived from the program specifications—this is the essence of the *functional approach* (or "black box" approach) to testing. But the claim is made[7] that to have a reasonable chance of constituting a reliable criterion, C must be composed of test predicates satisfying (at least) the following set of conditions:

Condition 1: Every individual branching condition in the program must be represented by an equivalent test predicate.

Condition 2: Every potential termination condition (e.g., error, overflow, etc.) must be represented by a corresponding test predicate.

Condition 3: The range of every variable appearing in a test predicate must be partitioned into classes that are "treated in the same way" by the program.

Condition 4: Every condition relevant to the proper functioning of the program that is implicit in the program specification or of one's knowledge of the program must be represented by a corresponding test predicate.

Condition 5: The test predicates must be "independent," in that all data satisfying a particular test predicate must exercise the same path in the program and must test the same branch conditions.

We note that only the second and fourth of these conditions are of a "functional" nature. The others are "structural," that is, relating more to the topology of the underlying flowchart. It would seem, therefore, that any reasonable testing strategy should address both points of view.

Consider the following example, the often cited problem of classifying triangles:

Specification:

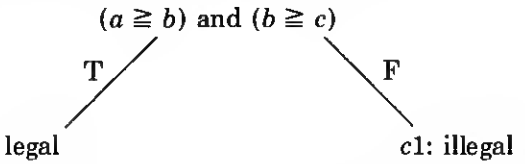
Input: Three positive integers $a \geqq b \geqq$ c.

Output: An indication as to whether:

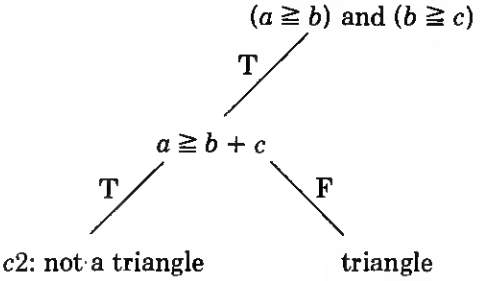
1. They do not represent the sides of a triangle
2. They are the sides of an equilateral triangle
3. They are the sides of an isosceles triangle
4. They are the sides of a scalene right triangle
5. They are the sides of a scalene obtuse triangle
6. They are the sides of a scalene acute triangle.

This problem is especially well suited to the "functional" approach.

Since the whole purpose of the problem is to classify its input domain, there is an obvious specification-based derivation of test predicates. We may first divide the universe of triples (*a*, *b*, c) into legal and illegal forms:



For the legal entries, we may further distinguish two cases:



and the triangles may then be subdivided into six subclasses:

$c3 : (a = b)$ and $(b = c)$ equilateral
$c4 : (a = b)$ and $(b > c)$ isosceles
$c5 : (a > b)$ and $(b = c)$ and $(a < b + c)$ isosceles
$c6 : (a > b)$ and $(b > c)$ and $(a*a = b*b + c*c)$ right scalene
$c7 : (a > b)$ and $(b > c)$ and $(a*a < b*b + c*c)$ acute scalene
$c8 : (a > b)$ and $(b > c)$ and $(a*a > b*b + c*c)$ and $(a < b + c)$ obtuse scalene.

If we set $C = \{c(i) : i = 1 \text{ to } 8\}$ and choose one triple from each input subdomain, we may obtain the test set:

$t1 = (1, 2, 3)$
$t2 = (14, 6, 4)$
$t3 = (1, 1, 1)$
$t4 = (2, 2, 1)$
$t5 = (3, 2, 2)$
t6 = (5, 4, 3)
$t7 = (6, 5, 4)$
$t8 = (4, 3, 2)$.

Such a test set will automatically satisfy (*) and the test selection criteria will more than likely meet Conditions 2 and 4 above. But we have no guarantee that the "structural" conditions 1, 3, 5 will be met, since we haven't looked at the program!

Weyuker and Ostrand[10,11] have made the cogent suggestion tbat the input domain be partitioned *both* on the basis of the specification-driven, program-independent properties mentioned above, and on the structural properties of the program as well. It seems tbat this is tbe only way to meet all five of the Goodenough and Gerhart conditions, and to thus have a cbance of approaching a reliable test selection criterion $C = \{c\}$ defined by a set of test predicates.

Suppose we add a sixth (implicit) condition to the five tbat are outlined above, namely:

Condition 6: The test predicates must be "complete" in that every input of the domain D must satisfy (exactly—see condition 5) one of the test predicates.

Then Conditions 5 and 6 ensure that $C = \{c\}$ defines a partition

$$\kappa = \{C\}$$

on the input domain. When we concentrate only on tbe problem specifications, as above, we obtain the *problem partition* consisting of *problem domains* C. Having a completed version of the program in band, we may speak as well of a *path partition*

$$\pi = \{P\}$$

of the same domain D, where each *path domain* P comprises a class of inputs that traverse the same path through the program. Thus, the path partition separates D into classes of inputs that are treated the same way by the program, whereas the problem partition separates D into classes that should be treated the same.

There is no assurance that these two partitions will coincide, nor is it necessary that they do. But ultimately (or at least, hopefully), the program and its algorithm all derive from the original problem specification, so we should not expect the two paritions to differ markedly. On the other hand, those differences that do exist are fruitful places to look for errors! Recognizing tbis, Weyuker and Ostrand bave suggested that the problem and path partitions be intersected, yielding a finer partition

$$\sigma = \kappa \wedge \pi = \{C \cap P\} = \{S\}$$

of nonoverlapping subdomains S of the domain D, and they further suggest that this be used as the ultimate test selection criterion, choosing one test case from eacb subdomain as before.

In the terminology of Weyuker and Ostrand,[10,11] a subdomain S of D is said to be *revealing* (of errors) if

$$\text{success}(s \text{ in } S) \Rightarrow \text{correct}(F, S),$$

i.e., if the successful execution of any input from S implies correctness

of the program over the whole subdomain. Since the inputs of a subdomain S (in the intersection above) should be and in fact are treated the same by the program, the hope is extended that these are, in essence, the revealing subdomains. A successful execution of one test datum from each of the subdomains S then implies (or at least suggests) the correctness of the program over the whole domain.

Consider the "triangle classification problem" once again, and suppose we are presented with the program (flowchart) of Fig. 1 as

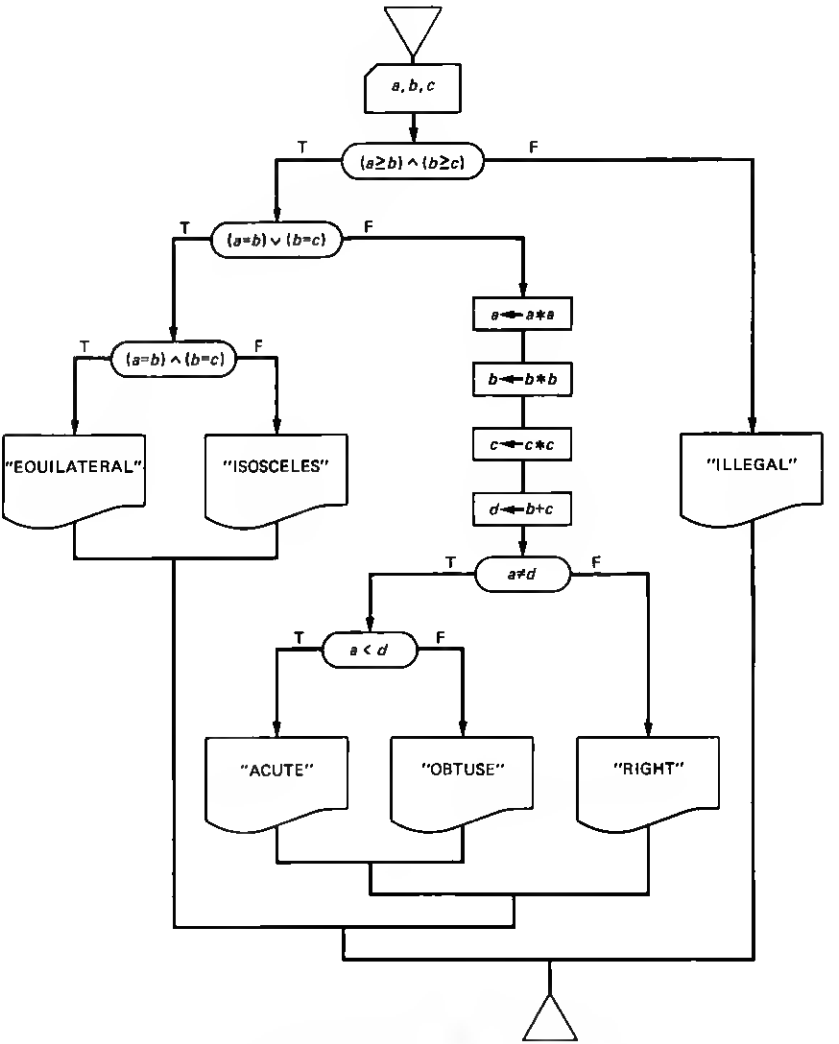


Fig. 1—Flowchart for classifying triangles.

representing a solution to the problem. There are six paths through tbe program, as described by the conjunctions of branch conditions defined by each path, as follows:

$p1 : \sim[(a \geqq b) \text{ and } (b \geqq c)] = (a < b) \text{ or } (b < c)$
$p2 : (a > b > c) \text{ and } (a*a = b*b + c*c)$
$p3 : (a > b > c) \text{ and } (a*a < b*b + c*c)$
$p4 : (a > b > c) \text{ and } (a*a > b*b + c*c)$
$p5 : (a \geqq b \geqq c) \text{ and } [(a = b) \text{ or } (b = c)] \text{ and } \sim[(a = b) \text{ and } (b = c)]$
$= (a = b > c) \text{ or } (a > b = c)$
$p6 : a = b = c.$

Intersecting the six corresponding path domains $P[i]$ ($i = 1$ to 6) with the eight earlier problem domains $C[i]$ results in a partition $\{S\}$ of nine subdomains characterized as follows:

$S1 = C1 = C1 \cap P1 : (a < b) \text{ or } (b < c)$
$S2 = C2 \cap P4 : (a > b > c) \text{ and } (a \geqq b + c)$
$S3 = C2 \cap P5 : (b = c) \text{ and } (a \geqq b + c)$
$S4 = C3 = C3 \cap P6 : a = b = c$
$S5 = C4 = C4 \cap P5 : a = b > c$
$S6 = C5 = C5 \cap P5 : (a > b = c) \text{ and } (a < b + c)$
$S7 = C6 = C6 \cap P2 : (a > b > c) \text{ and } (a*a = b*b + c*c)$
$S8 = C7 = C7 \cap P3 : (a > b > c) \text{ and } (a*a < b*b + c*c)$
$S9 = C8 = C8 \cap P4 : (a > b > c) \text{ and } (a*a > b*b + c*c)$
$\text{and } (a < b + c)$

in very close agreement with the problem partition $\{C\}$ obtained earlier.

The problem we are discussing has a rather precise functional specification so that we would expect that the problem and path partitions might nearly coincide. Nevertheless, there is a slight discrepancy, and in place of the test datum $t2 = (14, 6, 4)$ we would now have to choose two, say (14, 6, 4) and (3, 1, 1). A test of the resulting nine data points would then reveal two errors, as shown in Table I below.

Table I—Test of nine data points

Domain	Test Data	Correct Output	Actual Output
$S1$	(1, 2, 3)	Illegal	illegal
$S2$	(14, 6, 4)	Not a triangle	obtuse
$S3$	(3, 1, 1)	Not a triangle	isosceles
$S4$	(1, 1, 1)	Equilateral	equilateral
$S5$	(2, 2, 1)	Isosceles	isosceles
$S6$	(3, 2, 2)	Isosceles	isosceles
$S7$	(5, 4, 3)	Right	right
$S8$	(6, 5, 4)	Acute	acute
$S9$	(4, 3, 2)	Obtuse	obtuse

The programmer has failed to take account of those situations where $a \geqq b + c$ (not a triangle). And our test criteria are able to detect such errors. In fact, so detailed is the specification for this example that a test set based on the problem partition alone would have served equally well.

In spite of the obvious relevance of the ideas presented here, particularly those of Weyuker and Ostrand, a good deal of work remains to be done to apply the theory to a wide class of programs. One of the more important tasks is to find more systematic methods for constructing the problem partition. This will not be easy, since finding a good problem partition is quite similar to the task of creating the program itself. It is suggested, however, that the development of formal specification languages would be helpful here, particularly if such developments are made with a specification-driven testing methodology in mind, along the lines presented here.

An equally important consideration when thinking of applying the above theory to larger programs is that of obtaining the domains of the path partition. How are the paths to be described, generated, and selected with programs of increasing size and complexity? It is certainly clear that our one example is misleading in this respect. We had only a small number of paths to consider, whereas a typical program of any size will have a very large number of paths, most likely an infinity of paths, owing to the presence of loops. If our test is still to be finite, how do we then choose paths judiciously? How are they described? And most importantly, how do we generate test cases that will traverse these paths, if indeed this is possible? These are some of the questions that we begin to address in the following sections.

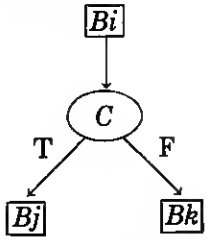
III. GENERAL THEORY—THE STRUCTURAL APPROACH

In a *structural approach* to the theory of testing, a program F is represented by a "skeleton" of its underlying flowchart, a directed graph symbolizing the flow of control. This point of view is advanced most effectively in the extremely lucid survey paper by Huang.[12] We should keep in mind, however, that the flowchart graph must be an accurate representation of the program flow in the code itself. Ordinarily, this is ensured through the use of a "tool" that automatically generates the flowgraph from the source program listing.

Using Huang's terminology,[12-14] a *program block* is a maximal sequence of program statements having the property that if the first member of the sequence is executed, then all other statements in the sequence will also be executed. A *program graph* $F = (V, E)$ is then a directed graph with vertex set V and edge set E, where each vertex is associated with a program block and in which there are pairs of edges:

(i, j) labeled by the condition C
(i, k) labeled by the condition $\sim C$

according as the flowchart segment:



encountered for blocks *Bi*, *Bj*, and *Bk*. (For convenience, we permit an empty hlock as a vertex in good standing, e.g., for treating an "if ··· then ···" statement with vacuous "else" clause.) It is further assumed that the graph has a single entry point, the *start vertex*, and a single exit point, the *stop vertex*, and that every vertex lies on some path from 'start' to 'stop.'

A *path* in a (program) graph is defined in the usual way, as a sequence of edges

$$p = e1, e2, \cdots, en,$$

though we ordinarily assume as well that we hegin the sequence at 'start' and end at 'stop.' Each such path has an associated *path predicate*

$$P = P1 \wedge \cdots \wedge Pn$$

written as a conjunction of the individual interpreted hranch condition lahels on the edges e, as discussed helow (see Section V). The path predicates P are to he identified in one-to-one correspondence with the path domains of the previous section. Thus we may write (somewhat amhiguously):

$$D = \cup P \quad \text{for} \quad P = \{d \text{ in } D : P(d)\}$$

so that the program function $F : D \to R$ is a union of functions $F(P) : P \to R$ restricting F to the individual path domains P.

In structured testing, we examine the program (as a digraph) and we seek to choose a finite set of paths that will cover the program with a certain degree of thoroughness. It is then hoped that test data causing the program to he successfully executed when traversing these paths are sufficient to warrant our confidence in the program's correctness. The theoretical underpinnings of such a testing plan have heen studied hy Howden,[15–18] who relates his work to the earlier study hy Goodenough and Gerhart.[7] Rather than speaking of a "test criteria," however, Howden refers to a *testing strategy*, as a uniform computahle function

$$(F : D \to R) \xrightarrow{H} (T, \text{ subset of } D)$$

that associates with each program F a finite test set T of D. H is said to be an *ideal strategy* if each $T = H(F)$ is an ideal test (for every program F). As is so often the case with testing theory, the first result is of a negative character:

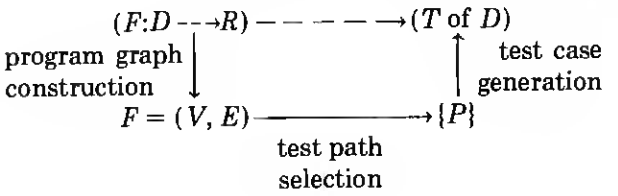
Theorem (Howden): An ideal testing strategy does not exist.

Nevertheless, Howden was able to show that "path testing" can be a reliable approach, at least for detecting certain types of errors. He takes the view that the program being treated is a member of a class of programs differing only as to whether they are correct, and for which the incorrect programs have errors of various (known) types. His objective was then to find, if possible, a restricted set of programs for which certain forms of structured testing (i.e., path testing) would be reliable. Typical of Howden's results is that which assumes that the error in a program does not change its control flow, i.e., that the set of path domains is not affected.

Theorem (Howden): Path testing is a reliable method for distinguishing correct from incorrect programs, as long as the errors of incorrect programs do not affect the path partition.

Of course, there are theoretical limitations in applying results such as this, since Howden has in mind our choosing one test datum from each path domain P, and these may be infinite in number. On the other hand, he has also devised a classification of error types that can be expected to lead to new insights into the testing problem generally. The reader should consult Howden's work (particularly Refs. 16 and 17) for further detail.

In a practical test setting, we require that the subset T of D be finite. Moreover, if we are speaking of a "path testing strategy," the above schema will be decomposed into the three-stage process,



summarized as follows:

1. Program graph construction
2. Test path selection
3. Test case generation.

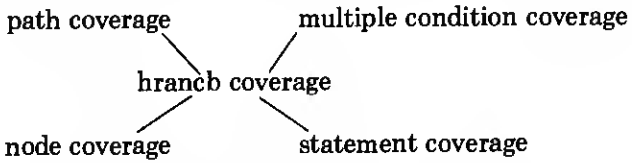
The first phase of the process, to construct the program graph from a source code listing, is fairly straightforward, and for most of the conventional programming languages, e.g., FORTRAN, Pascal, etc., such implementations are already in existence.

As a matter of fact, implementations of testing tools are in various stages of development for treating the entire process outlined ahove (e.g., see Clarke[19]). But, as we sball see, there are serious problems associated with the latter stages of any proposed implementation along tbese lines. Tbere are many alternative strategies to cboose from, and seemingly, none of these is hest for all situations. All we can do at this point is to outline the several alternatives and comment on their general suitahility. We begin hy introducing the various patb selection criteria, continuing tbis discussion in the next section. The last, and perhaps the most difficult, of our three subprocesses, the generation of test data, is treated in Section V.

There are, as we have indicated, a number of path selection criteria that can be used in attempting to devise a testing strategy that will provide a reasonable coverage of a program grapb. Among these criteria are:

1. *Statement coverage*: Execute all statements (hlocks) in the graph.
2. *Node coverage*: Encounter all decision node entry points in the graph.
3. *Branch coverage*: Encounter all exit hranches of each decision node in the graph.
4. *Multiple condition coverage*: Acbieve all possihle comhinations of condition outcomes at each decision node of the graph.
5. *Path coverage*: Traverse all paths in the graph.

These five strategies are related in their strength of coverage as shown below:



with the weaker criteria at the bottom and the stronger criteria at the top.

As an example illustrating the differing requirements of these criteria, consider the flowchart segment (program graph) shown in Fig. 2. In order to achieve node coverage, the single test:

$$abe\text{: } A = 2, B = 1, X = 1$$

will suffice (but it will not achieve statement coverage hecause the assignment $X \leftarrow X/A$ will not have heen executed). On the other hand, the single test:

$$ace\text{: } A = 2, B = 0, X = 3$$

will be sufficient for complete statement coverage (and node coverage

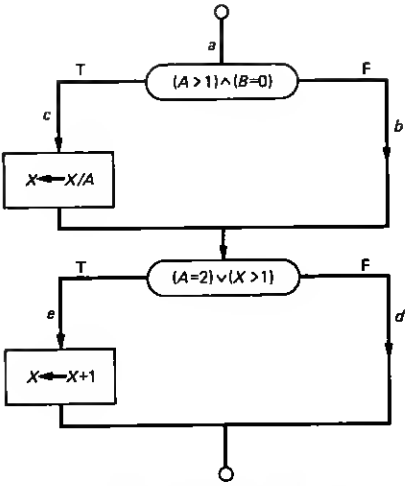


Fig. 2—Flowchart segment illustrating coverage criteria.

as well). For branch coverage, however, at least two tests would be required, e.g.,

$$acd\text{: } A = 3,\ B = 0,\ X = 3$$
$$abe\text{: } A - 2,\ B = 1,\ X = 1.$$

In the multiple condition coverage criterion, there are 2*2 + 2*2 or 8 outcomes to achieve in combination for the two simple conditions at each decision node. These may be satisfied, for example, by the selection of four separate tests, e.g.,

$$ace\text{: } A = 2,\ B = 0,\ X = 4$$
$$abe\text{: } A = 2,\ B = 1,\ X = 1$$
$$abe\text{: } A = 1,\ B = 0,\ X = 2$$
$$abd\text{: } A = 1,\ B = 1,\ X = 1.$$

The first test satisfies the conditions $A > 1, B = 0$ in the first decision and $A = 2, X > 1$ in the second decision. The second test ensures that $A > 1, B \neq 0$ for the first decision and $A = 2, X \leqq 1$ for the second decision. Further analysis shows that all eight combinations are achieved. For the path coverage criterion to be met, we again require four tests, e.g.,

$$ace\text{: } A = 2,\ B = 0,\ X = 4$$
$$acd\text{: } A = 3,\ B = 0,\ X = 3$$
$$abe\text{: } A = 1,\ B = 0,\ X = 2$$
$$abd\text{: } A = 1,\ B = 1,\ X = 1.$$

Note, however, that this test set would not satisfy the multiple condition coverage criterion.

It is clear that "statement coverage" and "node coverage" are in themselves rather weak strategies for testing, representing necessary hut hy no means sufficient criteria for a reasonahle structural test. The "hranch coverage" criterion, however, implies tbe other two (as seen in the diagram ahove) and has come to he regarded as a minimal standard of achievement in structure-based testing. Tbe stronger criteria of "multiple condition coverage" and "patb coverage" are difficult to achieve in a program of any complexity. In fact, the path testing criterion is usually relaxed to the extent that only "equivalence classes" of paths are represented. In a program of any size, particularly in the presence of program loops, there is a virtual infinity of paths through the program graph. Two paths are then considered "equivalent" if they differ only in their number of loop traversals. One then chooses only one representative from each such equivalence class in devising a test set. But still, this *modified path coverage* criterion is difficult to achieve in practice.

A survey of the literature shows that tbere is little common agreement as to what would be considered as an 'adequate' structural test criterion. As we have noted, the "branch coverage" criterion has heen widely recognized as a basic measure of testing thoroughness. This is evidenced hy the fact that most of the major software testing tools in existence or in development do indeed include some provision for achieving this particular test goal. The disagreement seems to he in deciding how much more (or less) is needed beyond this hasic requirement to entitle a structural testing strategy to be considered adequate.

If total branch coverage is indeed used as a measure of testing thoroughness, a simple calihration scheme can be invoked, using a set of *software counters*. One "prepares" the program for testing hy inserting counters at appropriate points in a modified copy of the program, and after running through the test set, one can determine the degree of thoroughness from a listing of the resulting counter values. This is the method of *test instrumentation*. We first define a *decision to decision* (DD) *path* of a program to he a sequence of a statements leading from a decision box (or tbe "start" node) to a decision box (or the "stop" node), having no intervening decisions. To determine whether every branch of our program has heen encountered at least once (branch coverage) in our testing, it is sufficient to insert a counter at the 'head' of each DD path.

Consider the classical flowchart solution (Fig. 3) to the problem of computing $z =$ "x to the power y". Here, there are five DD paths:

$$abc,\ d,\ efhi,\ gfhi,\ jk$$

and we therefore insert our software counters at the points a, d, e, g,

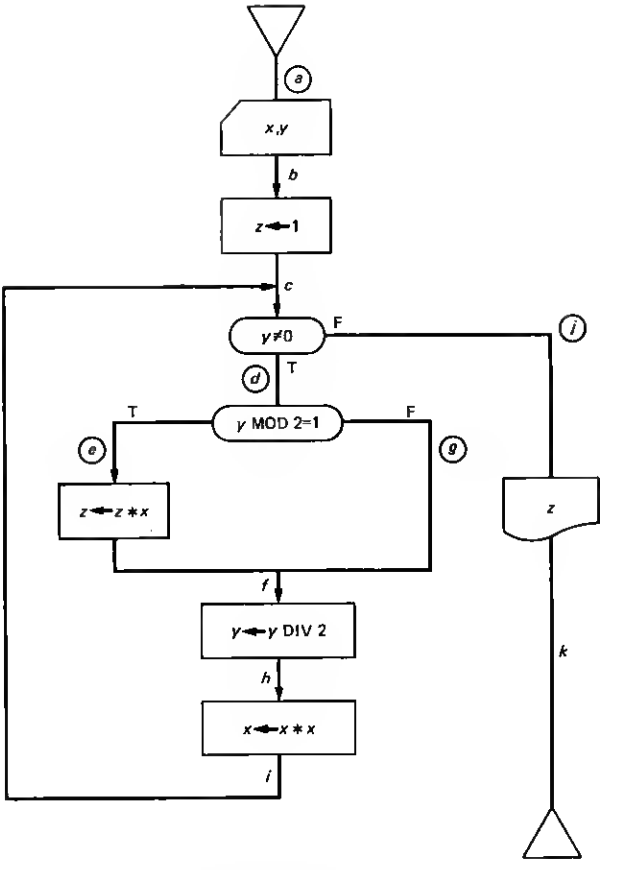


Fig. 3—Flowchart for computing x^y.

j, as shown. If we have run two test cases as shown in the table below,

x	y	a	d	e	g	j
10	0	1	0	0	0	1
20	1	1	1	1	0	1

we would find that 'counter g' has not yet heen activated. Inspection of the flowchart then shows that we need a test to traverse the path a, b, c, d, g, etc., requiring $y \neq 0$, $y \bmod 2 \neq 1$. So we may use $x = 24$ and $y = 2$, say, as an additional test case, thus ensuring complete branch coverage. Ideally, this latter phase, directing the tester to the area of untested code, would also he automated.

Of course, we would like to automate as much of the testing methodology as possihle, recalling the three-stage process mentioned earlier. On the other hand, in lieu of a complete mechanization, the testing

instrumentation scheme presented here can be of great help in isolating areas in need of further testing. Furthermore, it can be argued that for the little extra cost entailed, it is a worthwhile investment in any testing process, fully automated or not. [Parenthetically, we might note as an indication of the expense associated with the development of testing tools generally, that a package that does little more than "test instrumentation," as described here, has been announced recently (Computer, May 1982) by Management and Computer Services, selling for $12,000.00!.] It may be that some other criterion than "branch coverage" is being used as a measure of test thoroughness. It is still good practice to be concerned as to what extent this standard measure is being met. Moreover, it is reasonable to suppose that the "instrumentation concept," as exemplified here, might generalize to settings where other thoroughness criteria are being used.

IV. TEST PATH SELECTION

As we have indicated, there are a number of criteria that can be used in selecting program paths to achieve an adequate testing coverage. But the question then becomes: How do we automatically generate a collection of paths meeting a given criterion? The literature is somewhat "hazy" on this point. Perhaps the most explicit treatment of the problem is that of Paige,[20–22] in reference to programs built up from a strict adherence to the structured programming methodology. In fact, we know of no more general approach to the problem, one that would handle structured or unstructured code in relation to the whole spectrum of path selection criteria.

Recall that a *structured program* F is one that has been built up inductively from certain "simple statements" as a base (typically, the assignment, input and output statements, and procedure calls), using only the three familiar constructs:

1. *Sequence:* begin $P1$; $P2$; $\cdots$; Pn end
2. *Selection:* if C then P else Q
3. *Repetition:* while C do P

for structured (but possibly themselves compound) statements Pi, P, Q, respectively. The resulting program graph $F = (V, E)$ is then of a correspondingly restricted form, greatly facilitating the path analysis problem. Perhaps collapsing sequences of simple statements to a single block (graph node), one may then use a "regular expression" $r(F)$ to characterize the program flow, associating

1. The '$\cdot$' operator to sequences
2. The '+' operator to selections
3. The (Kleene) '*' operator to repetitions,

respectively.

For example, if F is the (structured) program graph shown in Fig.

4, we have

$$r(F) = a(b(d + e)(k + 1) + c(f + g(h(i + j))^*m))$$

as the corresponding regular expression. Note the loop $(h(i + j))^*$ resulting from a "while" statement.

We have mentioned earlier, in reference to the modified path coverage criterion, how an equivalence relation is often used to obtain a finite representation of the path alternatives in the presence of loops. Accordingly, if we make the substitution $x^* = x + 1$ (1 = null) in the regular expression $r(F)$, we acknowledge that a loop is either executed or not. Multiplying out so as to obtain a "sum of products" expression, one then obtains the desired collection of paths satisfying the modified path coverage criterion, e.g.,

abdk	*acf*
abdl	*acghim*
abek	*acghjm*
abel	*acgm*

in reference to the program graph of Fig. 4. On the other hand, it does not appear that this technique can be extended to handle unstructured programs.

But if we continue to deal with a structured program graph, we can describe a method for deriving a minimum number of paths sufficient to meet the "branch coverage" criterion. We assign a set of paths $S(r)$ to each regular expression $r = r(F)$ inductively, as follows. We let $S(a) = \{a\}$ for each simple statement a, and then, assuming that $S(r)$

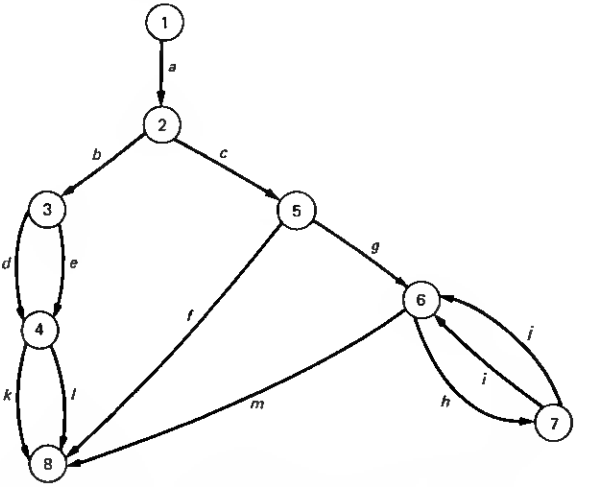


Fig. 4—Structured program graph illustrating modified path coverage criterion.

and $S(t)$ bave been defined, for regular expressions r and t, we set:

1. $S(r \cdot t) = S(r) \cdot S(t)$
2. $S(r + t) = S(r) \cup S(t)$
3. $S(r^*) = S(r)^*$.

Here, $S(r)^*$ is the singleton set obtained by concatenating (in any order) all of tbe paths in $S(r)$, and similarly, tbe set product $S(r) \cdot S(t)$ is obtained by concatenating paths in $S(r)$ with tbose in $S(t)$—but retaining only enough products so that each of the factors from $S(r)$ and $S(t)$ are represented. It follows that

1. $|S(r \cdot t)| = \max\{|S(r)|, |S(t)|\}$
2. $|S(r + t)| = |S(r)| + |S(t)|$
3. $|S(r^*)| = 1$.

By way of illustration, in considering once again the example from Fig. 4, we may compute:

$$S(i + j) = \{i, j\}$$
$$S(h(i + j)) = \{hi, hj\}$$
$$S((h(i + j))^*) = \{hihj\}$$
$$S(g(h(i + j))^*m) = \{ghihjm\},$$

etc., and finally,

$$S(r) = \{abdk, abel, acf, acghihjm\},$$

yielding four paths that together cover all of the branches of the program.

Once again, as in the case of the previous algorithm, there seems to be no easy extension of this technique that would handle unstructured programs as well. However, a general *upper bound* is readily available regarding the number of paths necessary for total branch coverage. Whether our program is structured or not, we make the observation that if a test path reaches a particular node of the program graph, then it must exit this node through one of tbe (two) branches leaving the node. If the grapb F has e edges and n nodes, it follows that

$$v(F) = e - (n - 2) = e - n + 2$$

is an upper bound on the number of paths necessary to achieve total branch coverage. Coincidentally, this is tbe formula for McCabe's *cyclomatic complexity measure*,[23] a figure that has proved to be useful in estimating overall "program complexity". At tbe same time, tbe graph theoretic derivation of a program's "independent" circuits (paths) yields a branch covering of paths, $v(F)$ in number—tbough generally somewbat in excess of tbe minimum number of paths that would be required. In tbe context of our running example, we have

$v(F) = 13 - 8 + 2 = 7$ and a corresponding set of *basis paths*:

acf *acgm*
abdk *acghim*
abek *acghjm*.
abel

Note that the single path *abdl* from our "path coverage" list that is not present here is itself a linear combination of patbs already listed. Note as well that we obtain seven paths here, whereas we know from the preceding analysis that four paths will suffice (for branch coverage).

The whole notion that McCabe's *basis of program paths* should constitute a goal of program testing has attracted considerable attention, and we feel obliged to comment on this point. Perhaps this is best done by listing what we think are the pros and cons to the approach. On the positive side, we cite the following:

1. The method is sufficiently general as to be applicable to both structured and unstructured programs.
2. The resulting "basis" does indeed ensure total branch coverage.
3. The paths of a basis are feasibly computable, using standard graph theoretic techniques.

On the other hand, these aspects must be counterbalanced with the following:

1. A single basis is not uniquely determined—there are many, and we must make a choice.
2. The number $v(F)$ of paths in a basis can greatly exceed the minimum number of paths required to achieve branch coverage.
3. The notion that, in some sense, *every* path in the program graph is accounted for by our having selected a basis is somewbat specious.

Note that we do not comment here on the inadequacies of McCabe's $v(F)$ as a measure of overall program complexity—we leave this discussion to a separate paper. On the other hand, the arguments for and against the use of the associated "basis of program patbs" as a testing strategy are inconclusive at best, particularly in comparison with the "level paths" of Paige[21,22] that we now describe.

In a program graph $F = (V, E)$, a *level-0 path* is a simple (acyclic) path from "start" to "stop". In effect, these paths trace the "fall through" conditions in the program. Then, inductively, a *level-i path* ($i > 0$) is a simple path (perhaps a circuit) that begins and ends on nodes of a path of lower level, but has none of its other nodes previously appearing on paths of a lower level. Intuitively, the level-i paths for $i > 0$ account for program loops of increasing nesting level and for feedback paths, etc., in the case of an unstructured program.

Considering our earlier structured program grapb (Fig. 4) and tbe

unstructured program graph of Fig. 5, we tabulate the respective level-i paths as shown in Tahles II and III.

In any case, we are ahle to say that a given level-$(i + 1)$ path is "associated with" a certain level-i path according as the given path hegins and ends on nodes of the parent path. This relationship orders the level paths in a tree-like structure, in such a way that one can readily construct test paths that again effect a total hranch coverage. In so doing, only level paths that associate can he comhined to form a program test path. Thus, for example, in the case of the program graph of Fig. 4 ahove, we may construct the path *acghihjm* as the linear comhination:

$$acghihjm = (6) + (7) + (8)$$

using the notation in Table II.

It is clear that the level paths of a program graph span the space of

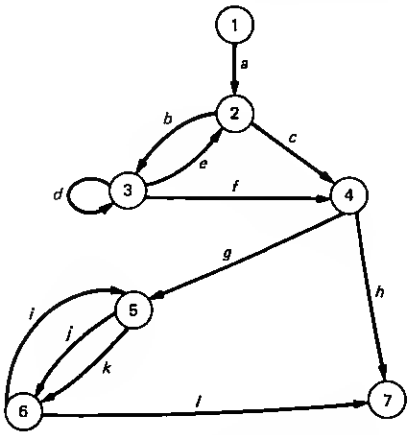


Fig. 5—Unstructured program graph.

Table II—Level-i paths for structured program graph (see Fig. 4)

Level	Level Paths	$V[i]$	$E[i]$
−1		{1, 8}	0
0	(1) *abdk* (2) *abdl* (3) *abek* (4) *abel* (5) *acf* (6) *acgm*	{2, 3, 4, 5, 6}	{*a*, *b*, *c*, *d*, *e*, *f*, *g*, *k*, *l*, *m*}
1	(7) *hi* (8) *hj*	{7}	{*h*, *i*, *j*}

Note: The sets $V[i]$ and $E[i]$ of vertices and edges at level-i are useful in the computation of the level-$(i + 1)$ paths.

Table III—Level-i paths for unstructured program graph (see Fig. 5)

Level	Level Paths	$V[i]$	$E[i]$
−1		{1, 7}	0
0	(1) *acgjl* (2) *acgkl* (3) *abfgjl* (4) *abfgkl* (5) *ach* (6) *abfh*	{2, 3, 4, 5, 6}	{*a*, *b*, *c*, *f*, *g*, *h*, *j*, *k*, *l*}
1	(7) *d* (8) *e* (9) *i*	0	{*d*, *e*, *i*}

Note: The sets $V[i]$ and $E[i]$ of vertices and edges at level-i are useful in the computation of the level-$(i + 1)$ paths.

program paths. But taken together, they do not usually constitute a basis. Thus, again in Fig. 4 above, we have eight-level paths, whereas we know from our previous analysis that this graph has rank $v = 7$. On the other hand, Paige's level paths bave a definite uniqueness, an advantage over tbe notion of a basis as developed by McCabe, and leading to a graduated *level path testing strategy* as follows:

1. First test all level-0 paths—in effect, keeping all loops in the "nonexecuting" mode.
2. Next test all level-1 paths, reaching them through tbeir associated level-0 paths, etc.

The result is a highly structured testing strategy wbere segments of the program are treated in successive layers of nesting depth.

The level path testing strategy provides for a rather exhaustive treatment of a program's path structure at successive depths of nesting. In this sense, tbe approach has a potential tboroughness rivaling tbat of the "modified path coverage" criterion. On tbe otber hand, Paige's strategy is readily applicable to both structured and unstructured programs. At the same time, his method lends itself to a convenient algorithmic solution,[22] though one must be prepared to compute all simple paths (or circuits) between various identified pairs of nodes, along edges not previously used—most likely requiring the use of a "depth first search" strategy. Except for this computational difficulty, the approach is quite orderly; it provides for a more tborough testing than simple branch coverage, and it compares favorably against McCabe's "basis of program paths" in that:

1. The level paths are uniquely determined.
2. The number of level paths will exceed $v(F)$.
3. The notion that somehow every path in tbe program graph is accounted for by our successive treatment of its levels has a good deal more credibility.

In conclusion, it must be noted that all of the methods we have discussed for selecting program test paths are subject to one overriding criticism. There is absolutely no assurance that the paths selected will be *feasible*, i.e., executable with an appropriate choice of input data. We suggest that this problem becomes more serious (and is surely more difficult to analyze) in the case of paths selected in an attempt to minimize the number required for branch coverage. Paige's "level path" strategy would seem to be easier to handle in this respect, since we build up paths from the simple to the more complex, starting with those that are more likely to be feasible.

V. TEST CASE GENERATION

The whole question of path feasibility is related to the "test case generation" problem. This is the one remaining phase to be discussed of the three that were outlined in the rectangular problem-decomposition paradigm of Section III. Considering the question of feasibility, however, we can see that it is difficult to so trichotomize the automation of the overall testing program. Though useful as a paradigm, we must admit that this partition of the problem is overly idealistic in relation to the real world of program testing that we are likely to encounter.

Suppose we have selected a set of program paths because they meet one or another of the test coverage criteria, or for whatever reason. There still remains the problem of generating corresponding test cases that will drive the program through the indicated paths. This again turns out to be a nontrivial (and in some cases, unsolvable) problem. All we can do at this point is to summarize the approaches that have been taken by researchers in the field and to give a few suggestions that might aid in developing a workable methodology.

Perhaps the most comprehensive treatment of the problem is that of Clarke.[19,24,25] Consider a single path p from "start" to "stop" through a program $F = (V, E)$, again viewed as a directed graph. We intend to show how p may be characterized as a *path predicate*, i.e., a logical condition

$$P = P1 \wedge P2 \wedge \cdots \wedge Pn$$

expressed as a conjunction of "interpreted" branch conditions derived from the labels on the edges of F. Inductively, we may think of p as having developed as a sequence of "partial paths:"

$$p(k) = (v[0] = \text{start}, v[1], \cdots, v[k])$$

leading from "start" to some intermediate vertex $v(k)$ on the way to

"stop". Correspondingly, we may give an inductive derivation of the path predicate, writing $P(0) = \text{true}$ and

$$P(k) = P(k-1) \wedge ibp(v[k-1], v[k]),$$

where the latter conjunct is the *interpreted branch predicate* associated with the transition from vertex $v[k-1]$ to $v[k]$. More precisely, $ibp(e)$ for an edge e laheled with the Boolean condition C will he computed hy suhstituting (in C) the current "symholic values" of all variahles according to their updating along the partial path $p(k)$.

For example, consider the flowchart solution (Fig. 6) for estimating the point where a function f takes on its maximum value. For the path

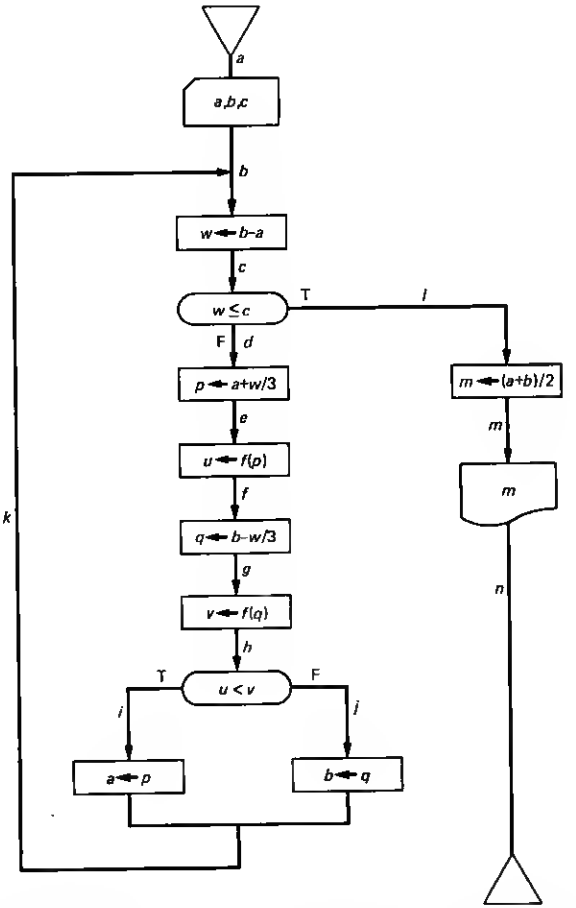


Fig. 6—Flowchart for estimating the point where a function is maximized.

$p = abclmn$, we compute

$$
\begin{aligned}
P(0) &= \text{true} \\
P(1) &= P(0) \wedge \text{true} = \text{true} \\
P(2) &= P(1) \wedge \text{true} = \text{true} \\
P(3) &= P(2) \wedge (b - a \leqq c) = (b - a \leqq c) \\
P(4) &= P(3) \wedge \text{true} = (b - a \leqq c)
\end{aligned}
$$

and finally,

$$P = P(5) = P(4) \wedge \text{true} = (b - a \leqq c),$$

noting that it was necessary to suhstitute $a - b$ for w in the condition for traversing edge 1 hecause of the earlier assignment statement.

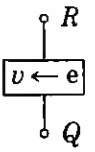
In general, this process of continually updating the symholic values of program variahles as we proceed along a path is called *symbolic execution* (or *symbolic evaluation*). The data descriptions generated in symholic execution provide a precise representation of the changing *program state*. Initially, the program state is the three-place vector:

$$
\begin{aligned}
\text{state} &= [\text{start, values (start), pathpred (start)}] \\
&= (\text{start, } (\perp, \perp, \cdots, \perp), \text{true}),
\end{aligned}
$$

where "values" tahulates the symholic values of all program variahles ($\perp$ = undefined), and "pathpred" stores the inductively generated path predicate P as described earlier. Symholic names are assigned (in "values") to input variahles whenever a read statement is encountered on the program path. Throughout the symholic evaluation, all symholic representations of variahle and hranch predicate values are then expressed in terms of these symholic names, as representatives of input values. In particular, as one encounters an assignment statement ($v \leftarrow$ e), the symholic value of the program variahle v is updated (as in the example ahove) through suhstitution of the symholic value of the expression e. In this way, "state" and especially the "values" vector will provide a continually updated snapshot of the program's develoopment along the path. Moreover, the final value of the "pathpred" component of "state" provides the logical conjunction described ahove.

This path predicate P defines a corresponding (path) subdomain of the input space D, and hy the nature of the symholic evaluation technique, P is expressed as a set of conditions on the input variahles alone. To generate a test case (of input data) that will cause the program to traverse the path p, it is then only necessary to find input values that satisfy all of these conditions. As we might expect, however, this is often easier said than done.

Before discussing this problem in any detail, it is better that we first describe an alternative to the above approach, one that proceeds in reverse—from the end of the path to its beginning. This technique, known appropriately as *backward substitution*, is best described in the survey paper by Huang.[12] To traverse a path p, certain conditions must be met, i.e., the set of branch conditions (C or $\sim C$) along the path must be satisfied as they are encountered. On the other hand, suppose that an assignment statement ($v \leftarrow$ e) intervenes, between "start" and the predicate Q, the latter representing a given branch condition (though modified by "partial backward substitution" as we are now describing). In the following flowchart segment:



if we want Q to be true after the assignment ($v \leftarrow$ e) has been executed, then the predicate $Q(v \leftarrow \text{e})$ must be satisfied prior to its execution. Here, $Q(v \leftarrow \text{e})$ is the predicate obtained by substituting the expression e for each occurrence of v in Q [and we speak of $Q(v \leftarrow \text{e})$ as the predicate obtained by *dragging* Q *backward* through the indicated assignment statement]. It follows that the conjunction

$$R \wedge Q(v \leftarrow \text{e})$$

is necessary for our passage along the edge with condition R (through the assignment) and then to satisfy Q.

Altogether, if we want the specific path p to be traversed in a program's execution, then we must drag each of its edge conditions backward to "start", and the conjunction of all resulting predicates must be satisfied by the corresponding test case of input data. Note once again that we obtain in this way a corresponding path predicate:

$$P = P1 \wedge P2 \wedge \cdots \wedge Pn,$$

i.e., a conjunction of modified branch conditions, each expressed in terms of the input variables to the program.

Consider once again the example of Fig. 6, and suppose we wish to traverse the path *abcdefghikclmn*. The listing shown below traces the dragging of the three necessary branch conditions backward along this

path:

l	$w \leqq c$		
c	$w \leqq c$		
k	$b - a \leqq c$		
i	$b - p \leqq c$	$u < v$	
h	$b - p \leqq c$	$u < v$	
g	$b - p \leqq c$	$u < f(q)$	
f	$b - p \leqq c$	$u < f(b - w/3)$	
e	$b - p \leqq c$	$f(p) < f(b - w/3)$	
d	$b - (a + w/3) \leqq c$	$f(a + w/3) < f(b - w/3)$	$\sim(w \leqq c)$
c	$b - (a + w/3) \leqq c$	$f(a + w/3) < f(b - w/3)$	$\sim(w \leqq c)$
b	$b - a - (b - a)/3 \leqq c$	$f(a + (b - a)/3) < f(b - (b - a)/3)$	$\sim(w \leqq c)$
a	$b - a - (b - a)/3 \leqq c$	$f(a + (b - a)/3) < f(b - (b - a)/3)$	$\sim(w \leqq c)$

One finally obtains the conjunction of three predicates:

$$P1 : b - a - (b - a)/3 \leqq c$$
$$P2 : f(a + (b - a)/3) < f(b - (b - a)/3)$$
$$P3 : \sim(b - a \leqq c)$$

all expressed in terms of the inputs a, b, c (and the "called" function f). For purposes of comparison, the reader may try to compute an equivalent predicate using the symbolic execution method.

In an overall comparison of these two methods, one can identify an obvious "trade-off." With backward substitution, we avoid the costly storage facility needed for the continuous updating of all the symbolic program variable values. On the other hand, an important advantage accrues to the symbolic execution method, one that is not available for the backward substitution technique. Namely, we are more easily able to determine whether a given path is (or will be) feasible. And we can make the determination early in the symbolic evaluation. We need only check that the inductively generated predicates $P(k)$ are noncontradictory, as far as they go. We begin with $P(0)$ = true—certainly there is no contradiction here. Then, in the expression for $P(k)$ in terms of $P(k - 1)$, we have only to see whether $ibp(v[k - 1], v[k])$ contradicts $P(k - 1)$. If so, $P(k)$ and hence P itself is contradictory, and the path p is infeasible. Otherwise, we keep going. Note, in comparison, that with the backward substitution method, we wouldn't know whether a path was feasible until all of the calculation (of P) was completed—a definite disadvantage.

One must note, however, that all such "logical satisfiability" problems as we are now beginning to consider are exceedingly difficult to handle in practice. We include here the satisfiability question that results from the use of the "backward substitution" technique or the forward "symbolic evaluation" method, whichever is used. At the

conclusion of the backward substitution, we have a system of constraints on the inputs to the problem, and unless these constraints can be "solved" for the input data, we don't have a test case at all. The same may be said for the forward symbolic execution, except for the slight advantage that we can be determining the satisfiability (or lack thereof) as we go.

Huang, in his survey paper,[12] presents a systematic approach for handling the satisfiability problem, and we now outline the major features of his plan. The simplifying assumption is made that the path predicate takes the form:

$$P = P1 \wedge P2 \wedge \cdots \wedge Pn,$$

where the Pi are nonnegated atomic expressions:

$$d\ R\ e$$

with d, e arithmetic expressions in the input variables and R one of the six relational operators: $<, \leqq, =, \langle\rangle, \geqq, >$. Such a system of atomic logical expressions can readily be rewritten in the *prenex normal form*:

$$(Ex1)(Ex2) \cdots (Exn)(x1 = e1) \wedge (x2 = e2) \wedge \cdots \wedge (xn = en),$$

where the E's are "existential quantifiers" on auxiliary variable x's, and the new expressions (the e's) are differences of d, e above, sufficient to transform any inequalities to equalities. The inequalities are, in effect, shifted to the auxiliary variables, thereby serving to normalize the solution space. Thus, in place of the inequality $2(b - a)/3 \leqq c$ at the end of the table above, we would have $(Ex1 \geqq 0)[x1 = c - 2(b - a)/3]$. Altogether, the three inequalities of that problem are similarly transformed, and we have instead the prenex normal form:

$$(Ex1 \geqq 0)(Ex2 \geqq 0)(Ex3 > 0)$$

$$\begin{aligned} x1 &= c - 2(b - a)/3 \\ x2 &= b + 2a - 6 \\ x3 &= b - a - c, \end{aligned}$$

one that is somewhat easier to handle.

From this point, standard techniques of linear algebra can be used to further transform the system into one where a minimum number of variables are involved. Thus, in the case of our running example, we can simplify the system so as to finally obtain:

$$\begin{aligned} &(Ex1 \geqq 0)(Ex2 \geqq 0)(Ex3 > 0) \\ &\quad 3x1 - x2 + 3x3 = 6 - 3a. \end{aligned}$$

From here, one may "guess" a solution, e.g., $x1 = x2 = 0$ and $x3 = 0.1$

say. One thereby obtains:

$$a = 1.9$$
$$b = 2.2$$
$$c = 0.2,$$

an input set that will cause the program to traverse the path *abcdefghikclmn* in Fig. 6, as originally required.

If we are going to have to "guess" a solution to the feasibility question in the end, bowever, the outrigbt "trial and error" approach of Ramamoorthy et al.[26] offers an attractive alternative. One makes the assumption, as before, tbat tbe path predicate P is expressed as a logical conjunction:

$$P = P1 \wedge P2 \wedge \cdots \wedge Pn,$$

where eacb of the Pi is a constraint on the program's input variables. Moreover, it is assumed that the input variables have been ordered as $v[1], v[2], \cdots, v[m]$. With each variable $v[i]$, we associate a set $S[i]$ of conjuncts from P, namely:

$$S[i] = \{Pj : \text{only } v[1], \cdots, v[i] \text{ occur in } Pj\}$$

and these sets are then used as the basis for the "trial and error" algorithm shown in Fig. 7.

Assuming tbat values have been found for $v[1], \cdots, v[i-1]$ satisfying all the conjuncts in $S[1], \cdots, S[i-1]$, we either solve for $v[i]$ or randomly choose $v[i]$, depending on whether the set $S[i]$ contains an equality relation in $v[i]$. We then substitute this value in the conjuncts of $S[i]$. Should we thereby arrive at a contradiction, we "backtrack" to the iteration $i-1$, generating a different value for $v[i-1]$. Otherwise, we go ahead to the iteration $i+1$. If the complete iteration on i concludes successfully, we arrive thereby at a "satisfiable" test case for the input variables of the program; otherwise we do not. Note that the "key" to the method is the fact that at each stage i, only the variable $v[i]$ has not yet been resolved. Note, however, that the loop at the right of Fig. 7 must include some criterion for deciding that "enough" random numbers have been tried in the current iteration. But however this is decided, it must be conceded that such an approach as presented here has much to offer in its favor, particularly considering the difficulty of the "satisfiability" question in general.

The authors[26] provide an example of the use of their algorithm on the "triangle classification problem" considered earlier. More generally, they suggest tbat the method has proven to be successful in treating a much wider class of problems. We would note further that the method could conceivably be applied to the "running satisfiability" questions that arise in the use of the "symbolic execution" technique.

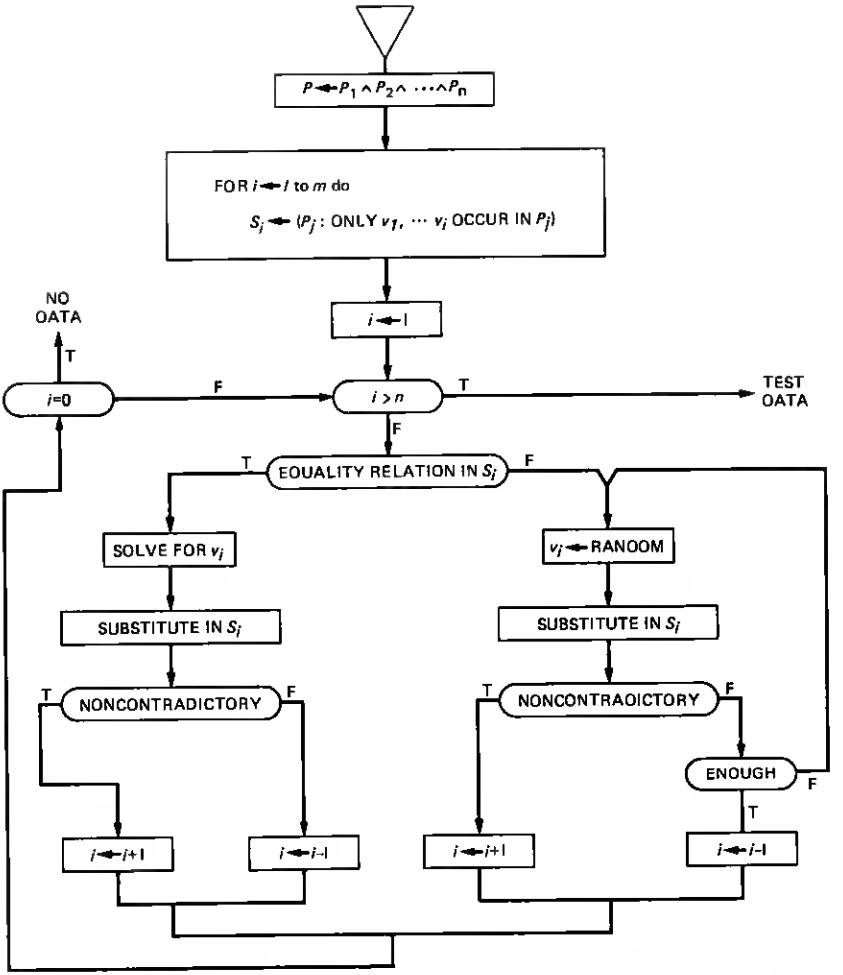


Fig. 7—Trial and error algorithm for solving the satisfiability problem.

In fact, it seems that this "trial and error" approach has a definite place—at least as a method of last resort, to be used as a component of any overall testing methodology.

Without some technique such as this, we are forced to rely on the extremely costly and not wholly reliable methods of "mathematical programming," particularly those routines that are designed to generate solutions to systems of inequalities. We cannot always assume that our systems are linear, in spite of the assumptions made by some authors. And in the absence of such an assumption, the problem is quite a difficult one, generally beyond the capability of the packages that are currently available.

Recognizing this, a most unusual and quite promising approach has been suggested by Kundu.[27] The idea is to combine tbe "test path selection" and "test case generation" phases of the solution, using the previous test case(s) $t[k]$ to help in determining the next test case $t[k+1]$. The result is a sequence of determinations:

$$(t[0]\rightarrow)p[0]\rightarrow t[1]\rightarrow p[1]\rightarrow \cdots$$

starting from an initial test case $t[0]$, chosen at random. The method is as follows:

1. Analyze $t[k]$: Execute the program with input $t[k]$, and determine its execution path $p[k]$. Then perform a (partial) symbolic execution of $p[k]$, so as to determine (an approximation to) its path predicate $P[k]$.
2. Select next test case: Determine the next test case $t[k+1]$ so that it violates at least one constraint in each of the path predicates $P[j]$, for $j < k$.

We are thus assured that each new test care $t[k+1]$ causes the program to traverse a genuinely new path, different from all those previously chosen.

In comparison with the previous methods we have discussed, Kundu reverses the roles of the test paths and the test data. The patb $p[k]$ is determined from $t[k]$ in order to guide the next test case $t[k+1]$ away from previous paths. That is, $p[k]$ is not used for finding an input that corresponds to that path itself. Therein lies the novelty of the approach.

Moreover, Kundu's method is definitely not designed with any specific measure of test thoroughness in mind. (He asserts that no good measures of testedness are available, anyway.) It is clear, however, that one could easily augment his procedure with test instrumentation devices, as discussed earlier, for the purpose of assuring that some standard test coverage criterion (e.g., branch coverage) has been met.

The primary advantage of Kundu's method is easily understood. Consider the constraint on $t[k+1]$ as described in (2) above, i.e.,

$$t[k+1] \text{ not in } P[1] \cup P[2] \cup \cdots \cup P[k].$$

It is clear that the "forbidden region" for $t[k+1]$ thus represents only a small portion of the total input domain D (see Fig. 8). This is so because the number of test cases generated in the testing activity is very small compared with the total number of executable paths in the program. Intuitively, the determination of the required $t[k+1]$ should thus be relatively easy. And for the same reason, the determination of a test datum in a given patb domain (as is required in the usual strategy) should be more difficult. Kundu (see Ref. 27, pp. 176–177)

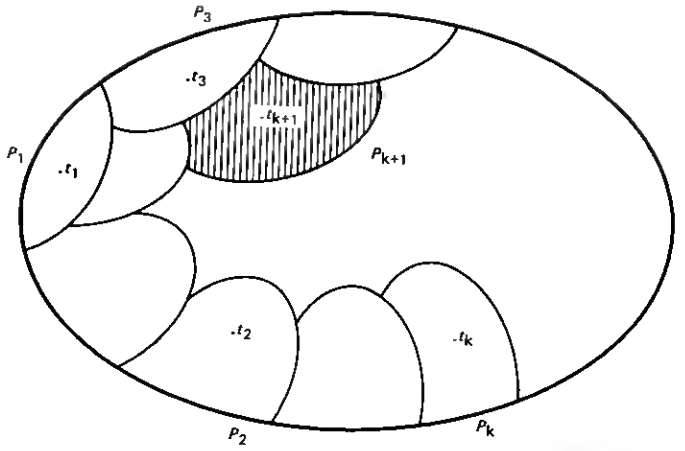


Fig. 8—Illustration of the forbidden region for selecting test cases.

gives a more detailed account of this reasoning, and the thrust of his argument is quite compelling. The reader may wish to consult Kundu's article for these additional details.

VI. CONCLUDING REMARKS

We have attempted to describe the many interesting and varied approaches to the program testing problem. Whereas no single approach to the problem may hold all the answers, it seems that there are enough good ideas around as to suggest the feasibility of a workable methodology, based on one or another combination of the strategies that have been advanced to date.

It must be remembered, however, that the thrust of our presentation, and, indeed, the main thrust in the literature has been toward the "unit test" level, where smaller programs are encountered. Thus, the ideas we have presented are, at the present time, feasible only in the case of programs of limited size. To think that we are nearing the point where we are ready to apply all of these techniques to the testing of an entire operating system or a compiler would be to miss the point completely. Nevertheless, our study has shown that indeed a start has been made.

We have tried to present a reasonably balanced survey of the recent contributions to the research literature on software testing methodology. It is perhaps likely that one or more worthwhile studies have escaped the author's attention, and therefore, their omission from this survey should not reflect on their importance to the development of the field. Moreover, the author can only hope that the studies that have been cited here have been presented in their best light. Limita-

tions of time and space have prevented a more complete treatment of these works, and for this, apologies to the authors are in order. At the same time, this author would like to acknowledge the use of the many cogent examples from the literature cited, hoping as well that these and other contributions have been faithfully reported.

In conclusion, the author would like to thank W. H. Leung, K. A. Gluck, and N. H. Petschenik for their most helpful comments in reviewing an earlier draft of the manuscript.

REFERENCES

1. W. C. Hetzel (ed.), *Program Test Methods*, Englewood Cliffs, NJ: Prentice Hall, 1973.
2. G. J. Myers, *The Art of Software Testing*, New York: John Wiley and Sons, 1979.
3. E. F. Miller, "Program Testing: Art Meets Theory," *Tutorial: Software Testing and Validation Techniques*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 390–8.
4. E. F. Miller, "Program Testing Technology in the 1980's," *Tutorial: Software Testing and Validation Techniques*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 399–406.
5. E. F. Miller, "Introduction to Software Testing Technology," *Tutorial: Software Testing and Validation Techniques*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 3–14.
6. E. F. Miller, M. R. Paige, J. P. Benson, and W. R. Wisehart, "Structural Techniques of Program Validation," *Tutorial: Software Testing and Validation Techniques*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 262–5.
7. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Trans. on Software Eng., *SE-1*, No. 2 (June 1975), pp. 156–73.
8. J. B. Goodenough, "A Survey of Program Testing Issues," in *Research Directions in Software Technology*, P. Wegner (ed.), Cambridge, MA: MIT Press, 1979, pp. 316–40.
9. R. G. Hamlet, "Test Reliability and Software Maintenance," Proc. Computer Software and Applications Conf. *COMPSAC 78*, November 13–16, 1978, Chicago, IL, New York: IEEE, 1978, pp. 315–20.
10. E. J. Weyuker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Trans. Software Eng., *SE-6*, No. 3 (May 1980), pp. 236–46.
11. E. J. Weyuker and T. J. Ostrand, "Current Directions in the Theory of Testing," Proc. Computer Software and Applications Conf., *COMPSAC 80*, October 27–31, 1980, Chicago, IL, New York: IEEE, 1980, pp. 386–9.
12. J. C. Huang, "An Approach to Program Testing," Computing Surveys, 7, No. 3 (September 1975), ppl. 114–28.
13. J. C. Huang, "Program Instrumentation and Software Testing," Computer, *11*, No. 4 (April 1978), pp. 25–32.
14. J. C. Huang, "Program Instrumentation: A Tool for Software Testing," *INFOTECH State of the Art Report, Software Testing*, Infotech Intl. Ltd. (1979), pp. 149–59.
15. W. E. Howden, "Methodology for the Generation of Program Test Data," IEEE Trans. Computers, *C-24*, No. 5 (May 1975), pp. 554–9.
16. W. E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Trans. Software Eng., *SE-2*, No. 3 (September 1976), pp. 208–15.
17. W. E. Howden, "Introduction to the Theory of Testing," in *Tutorial: Software Testing and Validation*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 16–19.
18. W. E. Howden, "A Survey of Dynamic Analysis Methods," in *Tutorial: Software Testing and Validation*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 184–206.
19. L. A. Clarke, "Automatic Test Data Selection Techniques," INFOTECH State of the Art Report, Software Testing, Infotech Intl. Ltd., 1979, pp. 43–63.
20. M. R. Paige, "On Sizing Software Testing for Structured Programs," Intl. Symp. on Fault Tolerant Computing, New York: IEEE, June 1977, p. 212.
21. M. R. Paige, "An Analytical Approach to Software Testing," Proc. Computer

Software and Applications Conf., *COMPSAC 78*, November 13–16, 1978, Chicago, IL, New York: IEEE, 1978, pp. 527–31.
22. M. R. Paige, "Program Graphs, an Algebra, and Their Implication for Programming," IEEE Trans. Software Eng., *SE-1*, No. 3 (Septemher 1975), pp. 286–91.
23. T. J. McCabe, "A Complexity Measure," IEEE Trans. Software Eng., *SE*-2, No. 4 (December 1976), pp. 308–19.
24. L. A. Clarke and D. J. Richardson, "Symbolic Evaluation Methods for Program Analysis," in *Program Flow Analysis*, Muchnick and Jones (eds.), Englewood Cliffs, NJ: Prentice Hall, 1981, pp. 264–300.
25. L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. Software Eng., *SE*-2, No. 3 (September 1976), pp. 215–22.
26. C. V. Ramamoorthy, S. B. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," IEEE Trans. Software Eng., *SE*-2, No. 4 (December 1976), pp. 293–300.
27. S. Kundu, "SETAR—A New Approach to Test Case Generation," INFOTECH State of the Art Report, Software Testing, Infotech Intl. Ltd., 1979, pp. 163–86.

AUTHOR

Ronald E. Prather, B.S. and M.S. (Electrical Engineering), 1955 and 1958, respectively; M.A. (Mathematics), 1966, University of California, Berkeley; Ph.D. (Mathematics), 1969, Syracuse University. Dr. Prather is a Professor of Mathematics and Computer Science at the University of Denver. He spent the 1982–1983 academic year on sahhatical leave with the Software Quality Analysis group at Bell Lahoratories in Denver. He is the author of *Discrete Mathematical Structures for Computer Science* (Houghton Mifflin, 1976) and *Problem Solving Principles: Programming With Pascal* (Prentice Hall, 1982).